Distributed File System

CSCI5105 PA3 Design Document

4/22/25

Matthew Breach and **Lily Hymes**

breac001 and **hymes019**

CONTENTS

1 DESIGN DESCRIPTION

1.1 REPLICA_SERVER.PY

replica_server.py is a hybrid thrift client/server program that manages information stored in the distributed file system. The system follows a quorum replication protocol using the Gifford algorithm. The system supports reading files, writing files and listing all currently stored files.

The program is split into the following internally called functions:

1. **main** - deals with parsing input arguments
2. **run_replica_server** - establishes the server as a server capable of handling RPC calls
3. **import_compute_nodes** - reads compute_nodes.txt to determine N_R, N_W and connection information to other servers
4. **setup_coordinator** - allows the designated coordinator to initialize itself
5. **open_client** - simplifies thrift RPC calls
6. **update_file_metadata** - updates version of stored file, or adds it if nonexisting
7. **dprint** - debug printing that only outputs if debug mode is set
8. **cord_read_files** - internal coordinator function, finds most recent version from read quorum
9. **cord_write_files** - internal coordinator function, finds most recent version from write quorum

The program is split into the follow externally called functions:

1. **get_all_files** - returns name and version number of all stored files
2. **get_version** - returns version of a specific file
3. **get_file_size** - returns size of a specific file
4. **request_data** - returns thrift binary data to send file data over RPC call
5. **copy_file** - informs a node to copy a file from another node
6. **cord_list_files** - called upon to coordinator, returns list of all files stored in each node
7. **insert_job** - allows a replica server to insert a job to the coordinator
8. **list_files** - called by client, allows client to receive list of all files
9. **read_file** - called by client, moves most recent version of file to its local storage
10. **write_file** - called by client, inserts a file to the dfs network
11. **finish_write** - informs coordinator to perform write phase 2
12. **finish_read** - informs coordinator read has finished

1.1.1 INITIALIZATION

When a replica server is initialized it configures its internal variables. It firsts reads the **compute_nodes.txt** file. This file contains information such as N_R, N_W and the list of connection information to all replica servers within the DFS network. This file also contains information on which server is the designated coordinator.

If the file determines itself is the coordinator it performs a few additional steps, notably initializing the system for queuing jobs.

1.1.2 JOB SCHEDULING

In this a quorum replication protocol DFS network, a coordinator is responsible for ensuring scheduled jobs operate sequentially. When a server calls inserts a job using the **insert_job** function, the individual thread is assigned a number that indicates its ordering of processing. The number assignment process is protected via a python thead library lock.

Once a number has been assigned the job waits for its turn. The coordinator server stores a global variable called **taskNumberProcessing** that indicates which jobs turn it is. After each operation is completed this variable is incremented. This allows for jobs to be run in the order in which processing order numbers are given out.

1.1.3 PROCESSING READS

When a client sends a read request to a replica server, which may not be the coordinator server, it first calls the **insert_job** function on the coordinator to insert a read request. This function will return connection information to the replica server indicating where to fetch the most recent file from.

On the coordinator side, once a read request is ready to be processed it generates a random set of N_R servers. It then queries these servers asking what version of the indicated file they store. The coordinator then returns the most up to date file and its severs connection information.

After the replica server receives connection information it uses the **copy_file** function to retrieve the file from the indicated server. It then updates its own stored information using the **update_file_metadata** function to indicate the file was transferred.

The server then calls the **finish_read** function on the coordinator, this informs the coordinator the job has been finished. It increments the **taskNumberProcessing** variable.

1.1.4 PROCESSING WRITES

Write requests are handled relatively similar to read requests. The replica server first queries the coordinator asking for the most up to date version of a given file. Likewise, it does this by calling the **insert_job** function, however this time indicating the request is a write.

The coordinator retrieves the newest version in a similar process to reads, however uses N_W instead of N_R as its quorum size.

Once the server receives information from the the coordinator it copies the given file into its internal file storage and updates its stored file information to reflect the move. The version of this file it set to the version returned from the coordinator plus one.

The server then informs the coordinator using the **finish_write** function. This function allows the coordinator to reuse its list of quorum participants and asks each one of them to copy the inserted file into their internal file storage. Each node will then update their internal information indicating the new version of the file. This function also increments the **taskNumberProcessing** variable to allow the next job to proceed.

1.1.5 LISTING FILES

Listing files is a much simpler procedure compared to reads or writes as it does not have to ensure concurrency with other operations.

When a client sends a file list request the server asks the coordinator to return a list of files. The coordinator then calls each server in its network and they return their internal stored file lists. The coordinator then combines all these lists and returns it to the replica_server which in turn returns it the client

The file list contains a list of server connection information and a list for each server containing their stored files names and versions

1.2 CLIENT.PY

client.py is a thrift client program that allows for a user to interact with the distributed file system by contacting any one of the replica servers.

The program is split into the following internally called functions:
1. **main** - deals with parsing input arguments
2. **open_client** - simplifies thrift RPC calls
3. **list_files** - lists all files within the DFS
4. **read_file** - transfer a given file to the local DFS replica
5. **write_file** - inserts a given file into the DFS network.

The client takes input through command line arguments and requests for the server to read a file, write a file, or list all currently stored files and their associated versions.

NOTE: The client doesn't actually *read* files, it simply copies the most reason version to the server it is attached to. The client could then theoretically read the data there.

2 TESTING METHODOLOGY

To evaluate the performance of our quorum-based replicated file system, we developed an automated benchmarking suite encapsulated in **test.py**. This script coordinates the full lifecycle of a test: it launches a configurable number of replica servers, initiates concurrent read/write requests from clients, logs all runtime results to a CSV file, and optionally generates visual heat-maps for analysis.

The testing methodology focuses on validating quroum behavior under various configurations and measuring the impact of different read/write quroum sizes (N_R, N_W) and client concurrency levels.

2.1 TEST AUTOMATION AND EXECUTION

Our test script supports fully automated experimentation using the following flow:

1. **Quorum Generation**
   - For each number of replica servers $n \in [7, \text{max\_nodes}]$, the script computes all valid (N_R, N_W) combinations that satisfy:

$$N_R + N_W > n \quad \text{and} \quad N_W > \frac{n}{2}$$

   - These combinations represent safe quorums per the quorum consistency protocol.
2. **Replica Launching**
   - **n** replica servers are spawned as subprocesses with their own storage directories.
   - The script dynamically assigns open TCP ports using Python's **socket** module and generates a **compute_nodes.txt** configuration file listing all node addresses and quorum parameters.
3. **Client Simulation**
   - For each valid (N_R, N_W) configuration and client count $c \in [3, \text{max\_clients}]$, the script launches c concurrent client threads.
   - Each thread performs either a **read** or **write** operation against the distributed system.
   - The system records the total time taken to complete all operations in that workload for the given configuration.
4. **Data Recording**
   - Results are saved in a structured CSV file (**results.csv**), with one row per test:

nodes, NR, NW, clients, read_s, write_s

- This ensures repeatable, analyzable results and avoids recomputations when not necessary.

2.2 HEAT-MAP GENERATION AND VISUALIZATION

To better understand how different quorum parameters and client counts affect performance, we generate read and write runtime heat-maps:

1. **Averaged Heat-maps**

   - For each unique quorum pair (N_R, N_W), we compute the average read and write time across all tested node/client configurations.
   - These are saved as:
     - **heatmaps/read_heatmap_avg.png**
     - **heatmaps/write_heatmap_avg.png**

2. **Per-Run Heat-maps**

   - We also generate one read/write heat-map per (n, c) pair to analyze how quorum performance scales with fixed node and client counts/
   - These are saved with filenames like **heatmaps/n9_c4_read.png**.

2.3 REPRODUCIBILITY AND MODULARITY

Our design emphasizes reproducibility:

- All test artifacts (temporary files etc..) are isolated in the **pa3_test/** directory, with node-specific folders for logs.
- The script includes a **--csv <file>** option to skip running new experiments and generate plots from prior data.
- The **--clean** option purges all test outputs (heat-maps, CSVs, node storage, and results file) after a confirmation prompt.

This methodology enables consistent, scalable testing without manual intervention and supports both batch experimentation and rapid prototyping of new protocol changes.

2.4 DISTRIBUTED TESTING

We also tested distributed operation across three cselabs machines (**csel-cuda-{01, 03, 04}.cselabs.umn.edu**) and verified DFS functionality by spinning up replica servers on each cuda system and performing a series of writes, reads, and lists to ensure proper functionality.

3 TESTING RESULTS

To evaluate the effects of quorum parameters on performance, we executed our testing suite with the following command:

python3 test.py 10 10 --plot

This exhaustively ran experiments on all valid combinations of N_R and N_W for replica counts from 7 to 10 and client counts from 3 to 10. The resulting read and write times were aggregated and visualized using heatmaps, shown below.

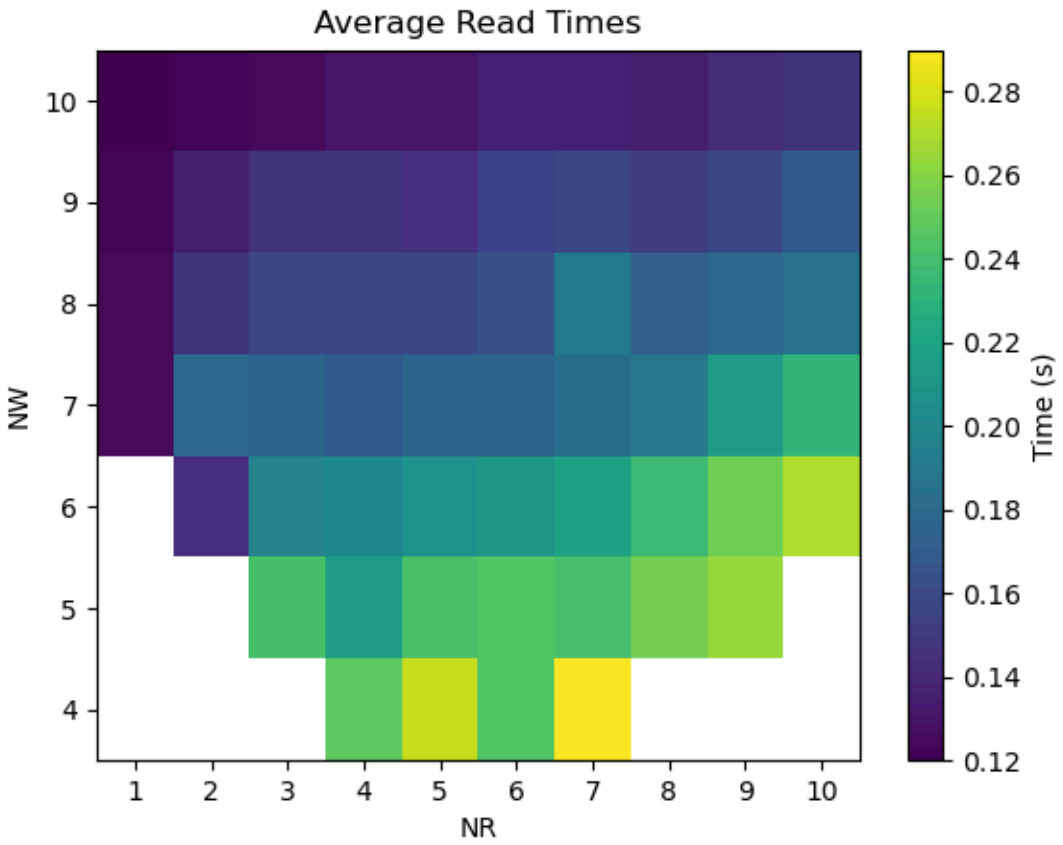

Figure 1: Average read latency (in seconds) across all valid quorum combinations. Color intensity increases with latency.

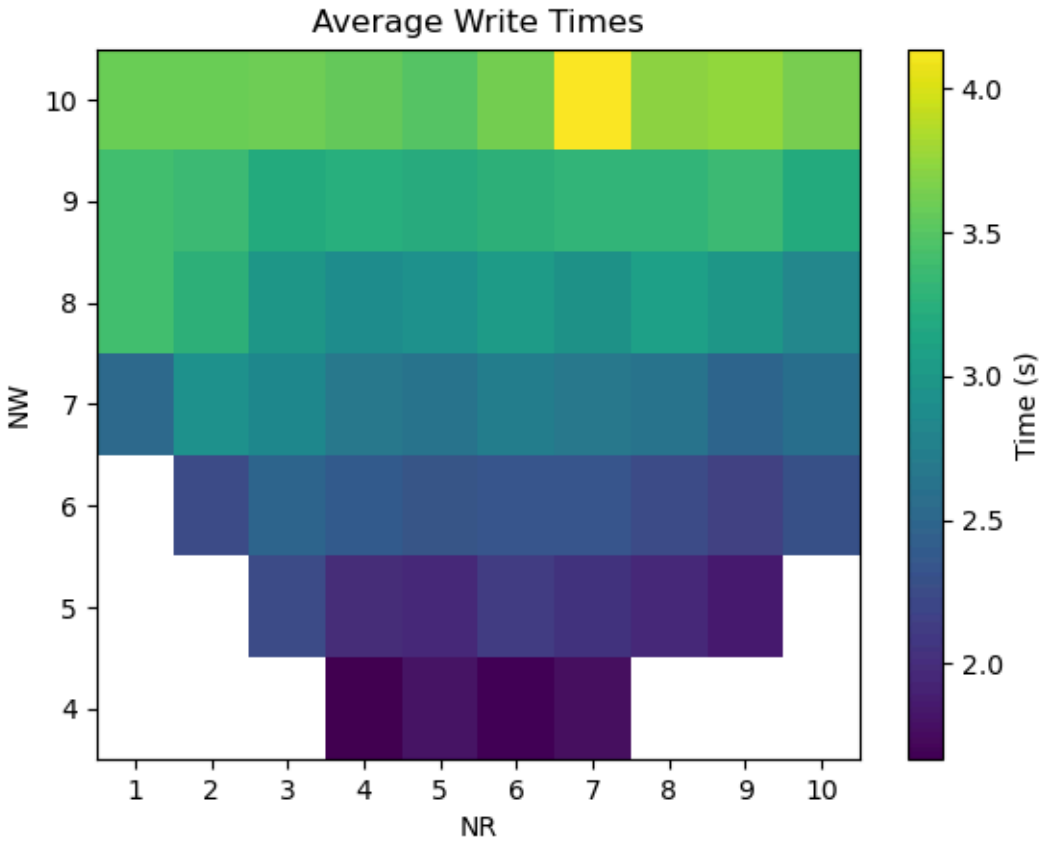

Figure 2: Average write latency (in seconds) across all valid quorum combinations. Color intensity increases with latency.

Note: Heatmaps for each combination of ***num_servers*** *and* ***num_clients*** *can be found here.*

3.1 READ PERFORMANCE

Read times were primarily influenced by the size of the write quorum (N_W), with higher values generally yielding faster reads. Surprisingly, the read quorum (N_R) had minimal impact overall—read performance within each (node, client) test remained relatively stable across different N_R values for a given N_W.

However, a subtle trend appears in the heatmap: for any fixed N_W, lower values of N_R tend to produce slightly better performance. This can be seen as a faint leftward color gradient across the rows of the heatmap. Additionally, there is a diagonal band of optimal performance—forming a sort of "ridge"—indicating that configurations above this ridge tend to perform better than those below.

3.2 WRITE PERFORMANCE

In contrast, write times were heavily influenced by N_W, with higher write quorums significantly increasing latency. This is expected since each write must propagate to and be acknowledged by N_W nodes.

Write latency is lowest when N_W is small, particularly when both N_R and N_W are near the minimum valid values. As N_W increases, latency rises sharply, forming a clear upward gradient in the heatmap.

3.3 OVERALL INTERPRETATION

Our results validate the theoretical expectations of quorum systems:

- Writes benefit from smaller N_W to reduce propagation and acknowledgment delay.
- Reads benefit from larger N_W, likely due to better consistency guarantees and reduced coordination.
- N_R plays a surprisingly minor role in both read and write latency, suggesting that most of the coordination overhead is governed by the write quorum size.

These results support the trade-off intrinsic to quorum designs: tuning N_R and N_W enables performance optimization based on read- or write-dominant workloads.

3.4 OPTIMAL QUORUM CONFIGURATIONS

The tables below describe the optimal values of N_R and N_W for each (**num_nods**, **num_clients**) pair for the test described above.

3.4.1 BEST READ QUORUM COMBINATIONS

nodes	clients	NR	NW	read time (s)	write time (s)
7	3	1	7	0.1	1.14
7	4	1	7	0.11	1.62
7	5	1	7	0.12	1.93
7	6	1	7	0.12	2.31
7	7	1	7	0.13	2.73
7	8	2	6	0.13	2.9
7	9	1	7	0.14	3.4
7	10	1	7	0.14	4.09
8	3	2	8	0.1	1.53
8	4	1	8	0.11	2.12
8	5	3	8	0.11	2.73
8	6	1	8	0.12	3.2
8	7	1	8	0.13	3.55
8	8	1	8	0.13	3.93
8	9	4	8	0.13	4.15
8	10	1	8	0.14	5.76
9	3	1	9	0.1	1.66
9	4	1	9	0.11	2.12

9	5	4	9	0.11	2.44
9	6	1	9	0.12	2.97
9	7	1	9	0.13	3.55
9	8	1	9	0.12	3.97
9	9	2	8	0.13	4.29
9	10	1	9	0.14	5.8
10	3	1	10	0.1	1.7
10	4	1	10	0.11	2.21
10	5	1	10	0.11	2.83
10	6	1	10	0.12	3.3
10	7	1	10	0.12	3.81
10	8	1	10	0.13	4.41
10	9	1	10	0.13	5.1
10	10	1	10	0.14	5.35

3.4.2 BEST WRITE QUORUM COMBINATIONS

nodes	clients	NR	NW	read time (s)	write time (s)
7	3	1	7	0.1	1.14
7	4	1	7	0.11	1.62
7	5	1	7	0.12	1.93
7	6	1	7	0.12	2.31
7	7	1	7	0.13	2.73
7	8	2	6	0.13	2.9
7	9	1	7	0.14	3.4
7	10	1	7	0.14	4.09
8	3	2	8	0.1	1.53
8	4	1	8	0.11	2.12
8	5	3	8	0.11	2.73
8	6	1	8	0.12	3.2
8	7	1	8	0.13	3.55
8	8	1	8	0.13	3.93
8	9	4	8	0.13	4.15
8	10	1	8	0.14	5.76
9	3	1	9	0.1	1.66
9	4	1	9	0.11	2.12
9	5	4	9	0.11	2.44
9	6	1	9	0.12	2.97

9	7	1	9	0.13	3.55
9	8	1	9	0.12	3.97
9	9	2	8	0.13	4.29
9	10	1	9	0.14	5.8
10	3	1	10	0.1	1.7
10	4	1	10	0.11	2.21
10	5	1	10	0.11	2.83
10	6	1	10	0.12	3.3
10	7	1	10	0.12	3.81
10	8	1	10	0.13	4.41
10	9	1	10	0.13	5.1
10	10	1	10	0.14	5.35